ARTIFICIAL INTELLIGENCE (CSC 462) LAB # 4

NAME:	MUAAZ BIN MUKHTAR
REG NO:	FA21-BSE-045
CLASS & SECTION:	BSSE-5A
SUBMITTED TO:	SIR WAQAS ALI
DATE SUBMITTED:	25-09-2023

Department of Computer Science

Lab Task 1:

Imagine going from Arad to Bucharest in the following map. Your goal is to minimize the distance mentioned in the map during your travel. Implement a depth first search to find the corresponding path.

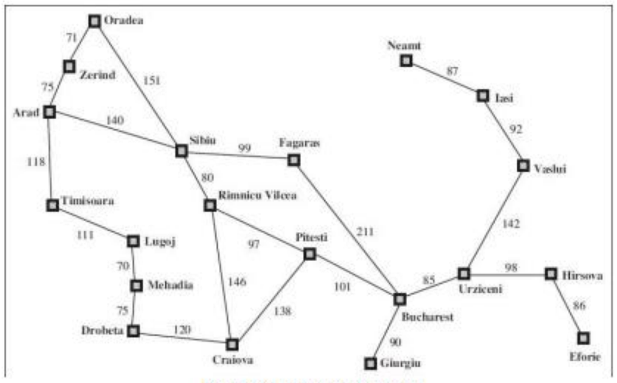
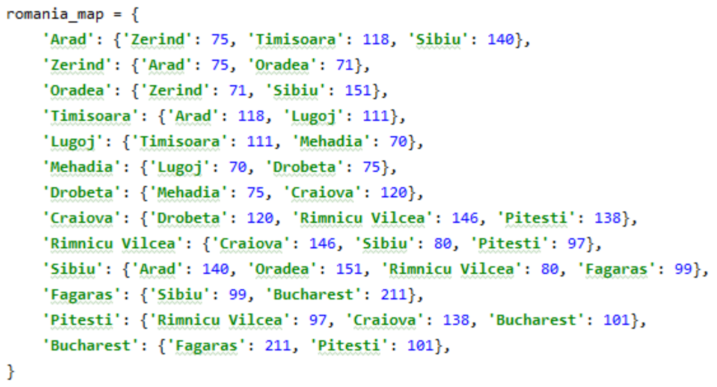
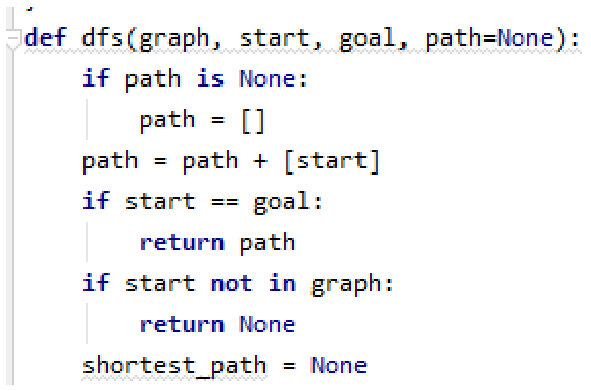


Figure 14 - Map of Romania

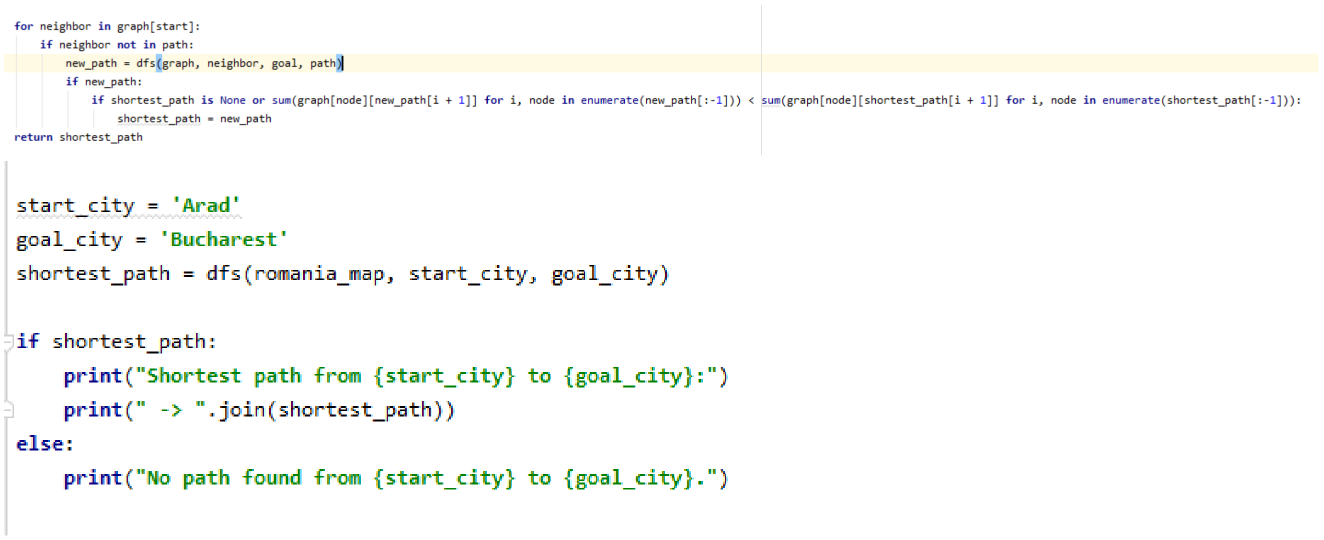
Code:



```
romania_map = {
    'Arad': {'Zerind': 75, 'Timisoara': 118, 'Sibiu': 140},
    'Zerind': {'Arad': 75, 'Oradea': 71},
    'Oradea': {'Zerind': 71, 'Sibiu': 151},
    'Timisoara': {'Arad': 118, 'Lugoj': 111},
    'Lugoj': {'Timisoara': 111, 'Mehadia': 70},
    'Mehadia': {'Lugoj': 70, 'Drobeta': 75},
    'Drobeta': {'Mehadia': 75, 'Craiova': 120},
    'Craiova': {'Drobeta': 120, 'Rimnicu Vilcea': 146, 'Pitesti': 138},
    'Rimnicu Vilcea': {'Craiova': 146, 'Sibiu': 80, 'Pitesti': 97},
    'Sibiu': {'Arad': 140, 'Oradea': 151, 'Rimnicu Vilcea': 80, 'Fagaras': 99},
    'Fagaras': {'Sibiu': 99, 'Bucharest': 211},
    'Pitesti': {'Rimnicu Vilcea': 97, 'Craiova': 138, 'Bucharest': 101},
    'Bucharest': {'Fagaras': 211, 'Pitesti': 101},
}
```

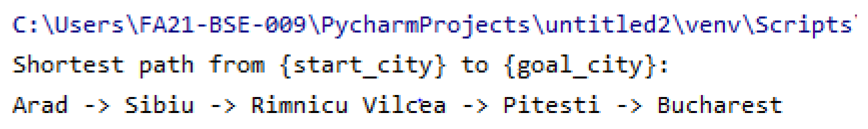


```
def dfs(graph, start, goal, path=None):
    if path is None:
        path = []
    path = path + [start]
    if start == goal:
        return path
    if start not in graph:
        return None
    shortest_path = None
```



```
for neighbor in graph[start]:
    if neighbor not in path:
        new_path = dfs(graph, neighbor, goal, path)
        if new_path:
            if shortest_path is None or sum(graph[node][new_path[i + 1]] for i, node in enumerate(new_path[:-1])) < sum(graph[node][shortest_path[i + 1]] for i, node in enumerate(shortest_path[:-1])):
                shortest_path = new_path
return shortest_path

start_city = 'Arad'
goal_city = 'Bucharest'
shortest_path = dfs(romania_map, start_city, goal_city)

if shortest_path:
    print("Shortest path from {start_city} to {goal_city}:")
    print(" -> ".join(shortest_path))
else:
    print("No path found from {start_city} to {goal_city}.")
```

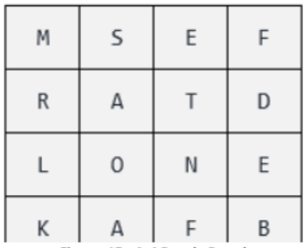
Output:



```
C:\Users\FA21-BSE-009\PycharmProjects\untitled2\venv\Scripts
Shortest path from {start_city} to {goal_city}:
Arad -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Bucharest
```

Lab Task 2:

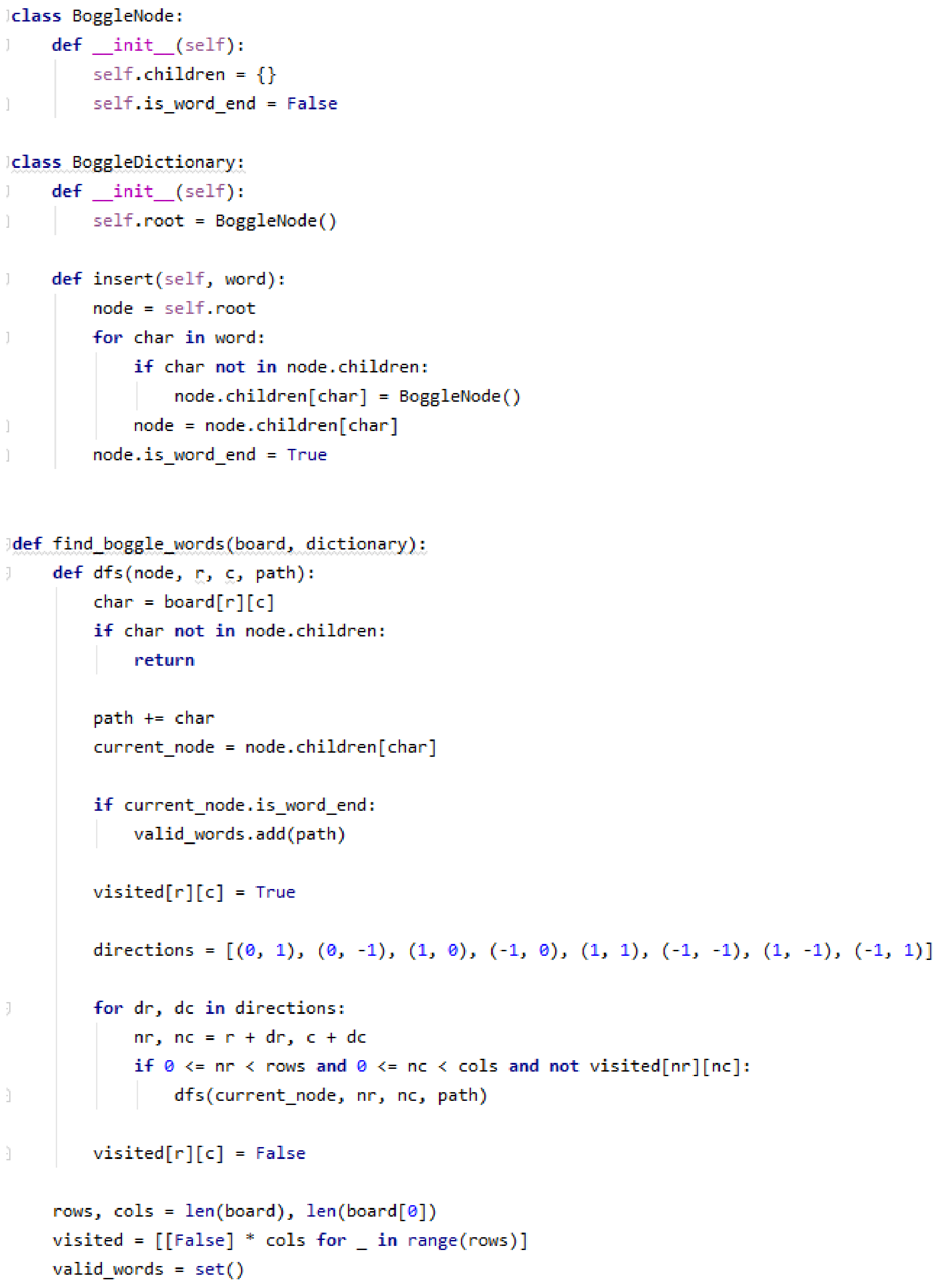
Generate a list of possible words from a character matrix Given an M × N boggle board, find a list of all possible words that can be formed by a sequence of adjacent characters on the board. We are allowed to search a word in all eight possible directions, i.e., North, West, South, East, NorthEast, North-West, South-East, South-West, but a word should not have multiple instances of the same cell. Consider the following the traditional 4 × 4 boggle board. If the input dictionary is [START, NOTE, SAND, STONED], the valid words are [NOTE, SAND, STONED].



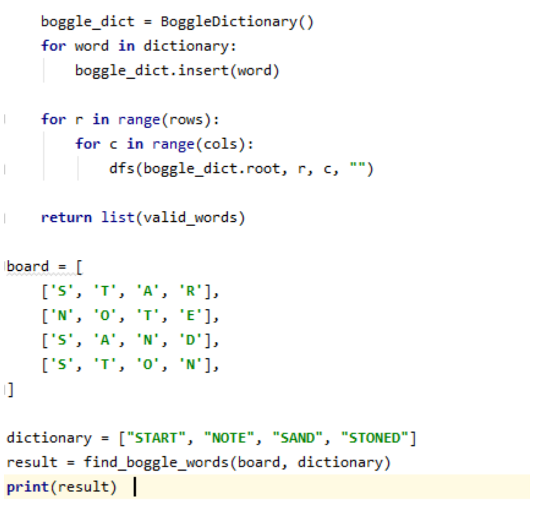
M	S	E	F
R	A	T	D
L	O	N	E
K	A	F	B

Figure 15 - 4x4 Boggle Board

Code:

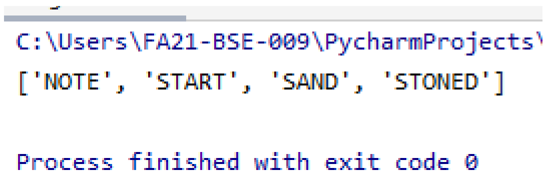


```python
class BoggleNode:
    def __init__(self):
        self.children = {}
        self.is_word_end = False

class BoggleDictionary:
    def __init__(self):
        self.root = BoggleNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = BoggleNode()
            node = node.children[char]
        node.is_word_end = True


def find_boggle_words(board, dictionary):
    def dfs(node, r, c, path):
        char = board[r][c]
        if char not in node.children:
            return

        path += char
        current_node = node.children[char]

        if current_node.is_word_end:
            valid_words.add(path)

        visited[r][c] = True

        directions = [(0, 1), (0, -1), (1, 0), (-1, 0), (1, 1), (-1, -1), (1, -1), (-1, 1)]

        for dr, dc in directions:
            nr, nc = r + dr, c + dc
            if 0 <= nr < rows and 0 <= nc < cols and not visited[nr][nc]:
                dfs(current_node, nr, nc, path)

        visited[r][c] = False

    rows, cols = len(board), len(board[0])
    visited = [[False] * cols for _ in range(rows)]
    valid_words = set()
```



```
    boggle_dict = BoggleDictionary()
    for word in dictionary:
        boggle_dict.insert(word)

    for r in range(rows):
        for c in range(cols):
            dfs(boggle_dict.root, r, c, "")

    return list(valid_words)

board = [
    ['S', 'T', 'A', 'R'],
    ['N', 'O', 'T', 'E'],
    ['S', 'A', 'N', 'D'],
    ['S', 'T', 'O', 'N'],
]

dictionary = ["START", "NOTE", "SAND", "STONED"]
result = find_boggle_words(board, dictionary)
print(result)
```

Output:



```
C:\Users\FA21-BSE-009\PycharmProjects\
['NOTE', 'START', 'SAND', 'STONED']

Process finished with exit code 0
```

.